Universidad Autónoma de Madrid
Escuela politécnica superior

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Trabajo Fin de Grado

DESARROLLO DE UN SISTEMA EFICIENTE DE ANÁLISIS DEL PROTOCOLO IEC 60870-5-104 PARA LA DETECCIÓN DE ANOMALÍAS EN REDES SCADA

Autor: David Sanches Gómez
Tutor: Jorge Enrique López de Vergara Méndez

JULIO 2019

DESARROLLO DE UN SISTEMA EFICIENTE DE ANÁLISIS DEL PROTOCOLO IEC 60870-5-104 PARA LA DETECCIÓN DE ANOMALÍAS EN REDES SCADA

Autor: David Sanches Gómez
Tutor: Jorge Enrique López de Vergara Méndez

High Performance Computing and Networking Research Group
Dpto. de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
JULIO 2019

Resumen

En los últimos años, el número de ataques hacia redes informáticas no ha hecho más que aumentar. La inclusión de controladores y dispositivos conectados dentro de la industria, ha supuesto la apertura a nuevas amenazas no contempladas hasta entonces, generando la necesidad de establecer medidas de seguridad.

Las redes SCADA, Supervisión, Control y Adquisición de Datos (*Supervisory Control and Data Acquisition*), suponen una estructura de vital importancia en el ámbito de los sistemas de control de datos. En la actualidad, se encuentran presentes en la mayoría de procesos industriales, como por ejemplo, las centrales eléctricas. En estos procesos, uno de los estándares usados es el IEC 60870-5. En este Trabajo Fin de Grado trataremos la extensión -104, utilizada de cara a mantener una mayor seguridad dentro de la red SCADA. No obstante, no está exenta de amenazas como se ha podido comprobar en los diferentes ataques ocurridos.

En este Trabajo Fin de Grado se ha desarrollado un sistema IDS (*Intrusion Detection System*) que se encarga de analizar los paquetes transportados por la red SCADA, a través de TCP/IP, de manera que permita la obtención de avisos ante posibles anomalías detectadas en la red. Este desarrollo parte de un Trabajo previo cuyo objetivo era el mismo, pero su desarrollo era en Python, motivo por el cual los tiempos de ejecución no eran adecuados para este tipo de sistemas. Por ello se ha realizado el detector en C para así poder disminuir dichos tiempos y poder detectar anomalías con gran velocidad.

También se han desarrollado las verificaciones necesarias que permiten validar el correcto funcionamiento del detector.

Palabras Clave

SCADA, IEC 60870-5-104, IDS

Abstract

In the last few years, the number of computer networks attacks has only increased. The inclusion of controllers and connected devices within the industry has meant the opening to new threats not contemplated until then, generating the need to establish security measures.

The SCADA networks (*Supervisory Control and Data Acquisition*) represent a structure of vital importance in the field of data control systems. Nowadays, they are present in the majority of industrial processes such as electric power plants. One of the standards that this processes use is IEC 60870-5. In this Final Degree Project we will deal with the extension -104, used to ensure greater security within the SCADA network, however it is not exempt from threats as it has been proven in the different attacks that have taken place over the last few years.

In this Project, an IDS(*Intrusion Detection System*) system has been developed. It is capable of analyze the packets transported by the SCADA network, through TCP/IP, allowing the reception of warnings facing possible anomalies detected on the network. This development starts from a previous work whose objective was the same, but its development was in Python, which is why the execution times were not suitable for these type of systems. For this reason, the detector has been coded in C in order to be able to reduce execution times and detect anomalies rapidly.

The different verifications necessary to validate the correct functioning of the detector will also be developed.

Keywords

SCADA, IEC 60870-5-104, IDS

Agradecimientos

Me gustaría comenzar agradeciendo a mi tutor, Jorge E. López de Vergara por haberme propuesto este tema, con el que he retomado mi atracción hacia las redes y a sus sistemas de seguridad. Gracias por el interés puesto y por el seguimiento continuado, el cual, ha supuesto una carga constante pero agradecida para la realización del trabajo. Sin duda, no podría haber tenido un tutor mejor.

De la misma manera agradecer a la Cátedra UAM-Naudit por facilitarme un ataque real, que ha sido de gran utilidad de cara a la validación del detector.

También me gustaría agradecer a mis padres, M.ª Eugenia y Antoine, y a mi novia, Ana, por todo el apoyo brindado, no solo durante el desarrollo de este trabajo, sino también a lo largo de los 4 años de carrera. Gracias por haberme inculcado la rutina de trabajo constante, sin la cual, todo lo conseguido no habría sido posible.

Agradecer también a mis compañeros de carrera, con los que he pasado grandes momentos, a la vez que hemos sufrido los numerosos exámenes a lo largo del grado. Me llevo grandes amistades de esta etapa de mi vida.

Por último, también agradecer a mi grupo de amigos de toda la vida, que han estado siempre ahí en todo momento y siempre lo estarán.

A todos, gracias.

David Sanches Gómez

Julio 2019

Índice general

Índice de Figuras

Índice de Tablas

Glosario de acrónimos

- **SCADA**: *Supervisory Control and Data Acquisition.*
- **IEC**: *International Electrotechnical Commission.*
- **IDS**: *Intrusion Detection System.*
- **TCP**: *Transmission Control Protocol.*
- **IP**: *Internet protocol.*
- **HMI**: *Human-MachineInterface.*
- **RTU**: *remote terminal unit.*
- **PLC**: *Programmable Logic Controller.*
- **IO**: *Dispositivos de entrada y salida.*
- **PAC**: *Programmable Automation Controller.*
- **VDF**: *Variable Frequency Drive.*
- **ISO**: *International organization of Standardization.*
- **OSI**: *Open System Interconnection.*
- **ASDU**: *Application Service Data Unit.*
- **APDU**: *Application Protocol Data Unit.*
- **APCI**: *Application Protocol Control Information.*
- **RTO**: *Retransmission timeout.*

1 Introducción

1.1. Motivación

La inclusión de las redes en la industria ha supuesto la apertura a nuevas amenazas. Estas, sin embargo, no afectan únicamente al software, sino que pueden ocasionar daños físicos significativos, modificando la funcionalidad de sistemas críticos y, por ende, estar en el punto de mira de cibercriminales y hacktivistas. Uno de los primeros ataques fue el Stuxnet **Figura 1.1**, el cual permitió ralentizar el proceso de enriquecimiento de uranio, alterando la velocidad de las centrifugadoras en las plantas nucleares iraníes, con el objetivo de frenar el programa armamentístico nuclear de ese país.

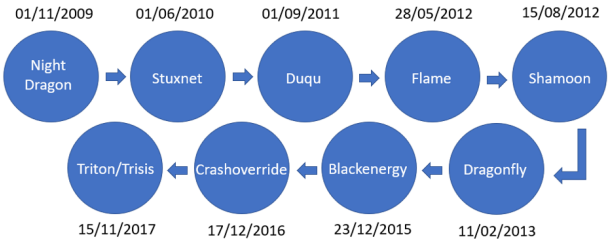


Figura 1.1: Historia de los ataques de tipo Scada.

El continuo aumento de dispositivos conectados, genera la necesidad de contar con herramientas de detección y protección hacia dichos ataques, los cuales, no harán más que aumentar ante la llegada de sistemas cada vez más críticos.

La realización del detector de anomalías, viene precedida de un Trabajo Fin de Grado previo[5]. Este se realizó en Python, motivo por el cual no pudo cumplir los requisitos de velocidad exigidos a la hora de detectar anomalías. Es por ello, que se propuso realizar este proyecto en C, un lenguaje con claros beneficios en cuanto energía consumida y velocidad de ejecución[6], como se pudo comprobar en la realización del trabajo.

1.2. Objetivos

El objetivo de este Trabajo Fin de Grado, ha sido el desarrollo de un sistema eficiente de análisis de tráfico anómalo en redes SCADA, Supervisión, Control y Adquisición de Datos (*Supervisory Control And Data Acquisition*), basados en el protocolo IEC 60870-5-104. Se pretende que el sistema IDS, Sistema de Detección de Intrusiones (*Intrusion Detection System*), analice el tráfico en tiempo real de una manera eficaz, ante la intrusión de paquetes anómalos. Como se verá más adelante en el capítulo **3**, es necesario que el sistema detector sea capaz de leer adecuadamente los campos de los paquetes trasmitidos. Posteriormente, se analizaron dichos campos verificando las posibles anomalías, para mostrarlas por pantalla en el caso en el que se detecten. Otro objetivo de gran importancia es la reducción de tiempos. Se requiere, que este detector realice la ejecución en el mínimo tiempo posible.

1.3. Fases de realización

La realización de este trabajo se ha dividido en las siguiente etapas, como se puede ver en el Diagrama de Gantt de la **Figura 1.2**.

- **Estudio del estado del arte:** Durante esta etapa, se realizó la lectura de diversas publicaciones, para así conocer la implantación de la redes SCADA. En particular, se indagó en el protocolo IEC 60870-5-104, estudiando su implementación y estructura, que posteriormente se explicará.
- **Diseño y Desarrollo:** En esta fase, se implementó el programa que se encargará de analizar el tráfico de la red, el cual es transportado a través de TCP/IP, en busca de anomalías. En ella se realizaron ciertas mejoras para optimizar el código, analizando los tiempos de consumo de cada función del programa, con la finalidad de reducir los tiempos de ejecución.
- **Validación y pruebas:** En ella se llevó a cabo la verificación del código, observando su correcto funcionamiento una vez subsanado los errores detectados. También, se realizó una serie de pruebas en entornos provocados, de manera que se pudiera verificar los detectores programados durante la fase de desarrollo.
- **Escritura de la memoria:** Por último, se redactó el presente documento, para así plasmar el aprendizaje obtenido durante el trabajo y los datos obtenidos para que, en un futuro, se pueda proseguir y añadir nuevas funcionalidades al programa detector.

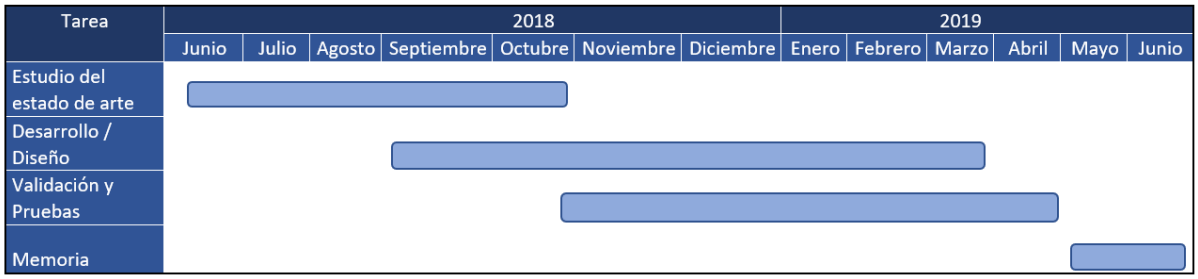


Figura 1.2: Diagrama de Gantt.

1.4. Estructura del documento

El resto de la memoria consta de los siguientes capítulos:

- **Estado del Arte:** En esta primera fase se llevará a cabo la explicación de las redes SCADA, que trataremos en este trabajo y desglosaremos en profundidad el protocolo en el que nos centramos, el IEC 60870-5-104, en el que explicaremos su estructura y los posibles ataques que puede recibir el mismo.
- **Análisis de Requisitos:** Una vez finalizada el desarrollo del estado del arte, nos centraremos en explicar las funcionalidades y resultados que se esperan de este Trabajo Fin de Grado. Reflejándose mediante requisitos funcionales y no funcionales, todas las exigencias que se desean que se cumplan.
- **Desarrollo y Diseño:** En esta fase se explicarán los pasos seguidos para la realización del detector. Se comentará las medidas implementadas para la correcta obtención de los resultados y comprobación de anomalías. También, se hablará de las mejoras llevadas a cabo para optimizar el programa, así como, la descripción del entorno usado para la realización del trabajo.
- **Validación:** Tras la descripción del desarrollo realizado, se proseguirá a detallar el proceso de validación llevado a cabo, especificando las diferentes pruebas realizadas y los resultados obtenidos en ellas. También, se detallarán las anomalías detectadas en las pruebas efectuadas.
- **Conclusiones y Trabajo Futuro:** En este apartado, se detallarán las conclusiones finales extraídas tras la finalización del proyecto, así como, las posibles modificaciones o trabajos futuros, que se pudieran realizar a partir de este.

2

Estado del Arte

2.1. Introducción

A lo largo de este capitulo, se expondrán los conocimientos previos obtenidos para la realización del proyecto. Servirá también de ayuda al lector, para así poseer contexto y conocimiento de los temas tratados en este Trabajo Fin de Grado.

Se hará una descripción de las redes SCADA, así como del protocolo IEC 60870-5-104, desglosando toda su estructura. También se hará mención de un trabajo previo, que sirvió como guía y objetivo a batir para este trabajo, así como de las medidas de detección.

2.2. SCADA

SCADA, *Supervisory Control and Data Acquisition* cuyas siglas en español significan Supervisión, Control y Adquisición de Datos, son sistemas usados para supervisar, controlar y analizar procesos en un amplio rango de industrias, como pueden ser fábricas, centrales de tratamiento de aguas, centrales eléctricas, entre otras.

Un sistema SCADA comienza con la comunicación, en tiempo real, con los controladores que se encuentran realizando el proceso en sí. Normalmente, estos controladores son PLC o RTU, de los que hablaremos posteriormente. El sistema SCADA, obtendrá información de estos controladores integrándolos todos en el sistema. De esta manera, se presentará utilizando una interfaz gráfica de usuario, a los operarios (HMI[1]), permitiendo que puedan comprobar, en tiempo real, que es lo que está haciendo el proceso y puedan reaccionar a alarmas, controlar el proceso, cambiar parámetros, entre otros.

Adicionalmente, se suelen incluir a estos sistemas un historial de manera que permita guardar la información en tiempo real, así como generar gráficos y estadísticas que permitan realizar predicciones futuras.

[1] Human-Machine Interface.

Un ejemplo de arquitectura de red SCADA podría ser el de la **Figura 2.1**.

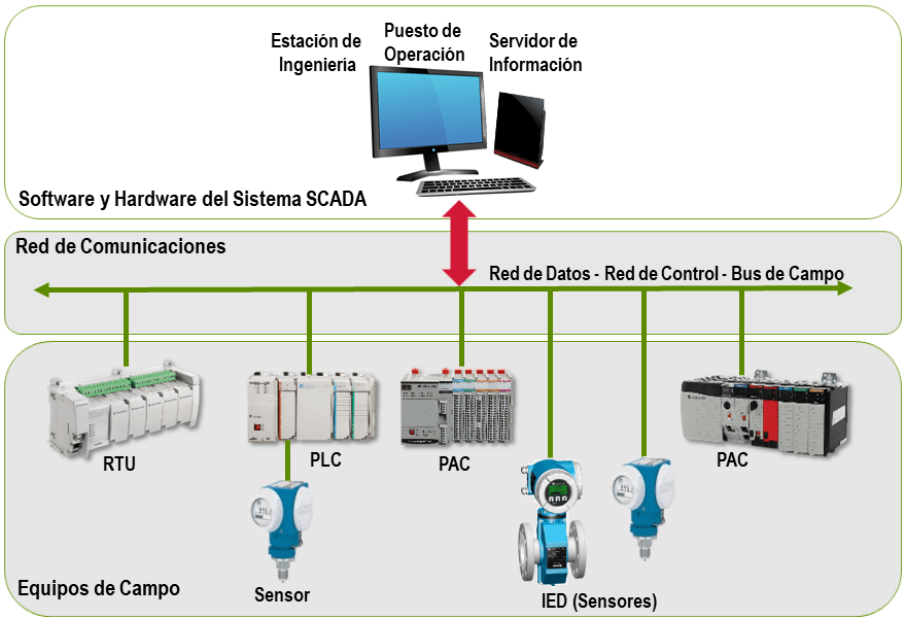


Figura 2.1: Arquitectura Básica de un Sistema SCADA [1].

A continuación, se detallarán brevemente los dispositivos que aparecen en la arquitectura básica de un sistema SCADA presentada en la **Figura 2.1**.

RTU

Cuyas siglas traducidas del inglés significan Unidad Terminal Remota (*Remote Terminal Unit*), es un dispositivo electrónico que se encarga de tomar los datos obtenidos por dispositivos IO[2], para luego enviarlos al controlador.

Este dispositivo, dentro de la red SCADA, hará la función de esclavo que espera ordenes de un maestro, que será la extracción de los datos.

PLC

La función de los Controladores Lógicos Programables (*Programmable Logic Controllers*) es muy similar a los RTU, es decir, la obtención de información en tiempo real, a partir de los IO del sistema. Tienen la capacidad de manejar entradas y salidas analógicas.

En la actualidad, los sistemas SCADA cuentan con un mayor número de PLC debido a su precio más económico, y al igual que los RTU, realizan la función de esclavo dentro de la red SCADA.

[2]Dispositivos de entrada y salida.

PAC

Los PAC, Controladores de Automatización Programables (*Programmable Automation Controller*), son dispositivos muy parecidos a los PLC, con los cuales comparten funciones, pero añaden otras adicionales [7]:

- Manejo de servomotores.
- Procesamiento avanzado para señales analógicas.
- Comunicaciones avanzadas, como gateways entre redes.
- Conjunto de instrucciones para manejo de VDF[3].

2.3. Estándar IEC 60870-5-104

El protocolo IEC 60870-5-104, es un estándar internacional lanzado por la IEC[4] en el año 2000. Como indica en la página Ipcomm [8], este está basado en el estándar IEC 60870-5-101, con el que comparte misma estructura en la capa de aplicación.

Este protocolo se encarga de establecer comunicación vía TCP/IP, para garantizar una transmisión de datos segura, entre el centro de control del sistema SCADA y los distintos controladores, los cuales hemos descrito previamente. También, establece que el puerto general de uso para las transmisiones TCP es el 2404.

Este puerto, corresponde a aquellos paquetes considerados como esclavos. Estos esclavos, solo pueden enviar respuestas o información ante una pregunta de un maestro, que será el encargado de dar las ordenes o recopilar la información proporcionada por los esclavos. Un ejemplo de esta arquitectura se puede ver en la **Figura 2.2**.

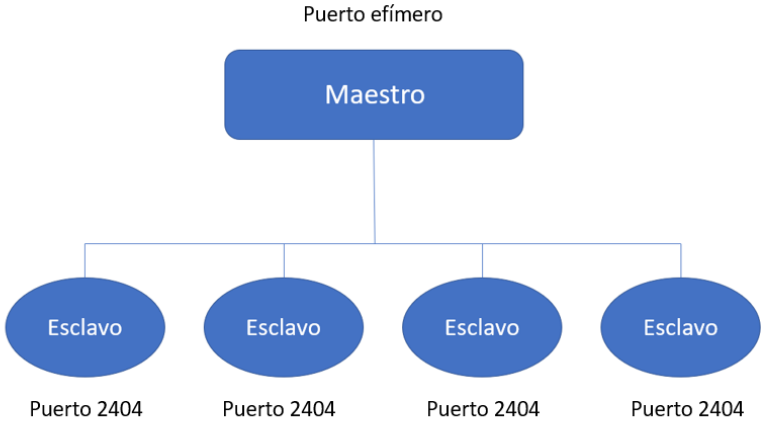


Figura 2.2: Arquitectura Maestro-Esclavo IEC 60870-5-104.

[3]Variadores de frecuencia.
[4]International Electrotechnical Commission.

Uno de los motivos por los que se usa este protocolo es que permite la comunicación a través de una red estándar, lo que posibilita transmisiones de datos simultáneas entre los distintos dispositivos y servicios.

Al igual que el IEC 60870-5-101, cumple el modelo ISO/OSI[5] el cual es un modelo con arquitectura de 7 capas como se ve en la **Tabla 2.1**).

7. Aplicación **6. Presentación** **5. Sesión**	APDU
4. Transporte	TCP
3. Red	IP
2. Enlace	Transmisión de Ip por ethernet
1. Física	Etheret

Tabla 2.1: Modelo ISO/OSI para IEC 60870-5-104.

Estructura

En la **Figura 2.3**, se muestra el formato con el que cuentan las tramas que usan el protocolo en cuestión. Esta cabecera recibe el nombre de ASDU, Unidades de Datos de Servicios de Aplicación (*Application Service Data Unit*), y todas ellas cuentan con dos campos, el campo START cuyo valor, siempre fijo, es 64h[6] y el campo que establece la longitud de la cabecera APDU, Unidades de Datos de Protocolos de Aplicación (*Application Protocol Data Units).*

Le siguen 4 octetos de control, que irán variando según el tipo PDU[7] del paquete, como vemos en la **Figura 2.4**

Para distinguir el tipo de trama, en la cabecera APCI, Información de Control de Protocolo de Aplicación(*Application Protocol Control Information*), basta con fijarnos en el primer octeto. En él, los dos últimos bits indican que tipo de trama es:

- **X0:** Para tramas de tipo I, *numbered information transfer*. Se usa para realizar distintas funciones de transmisión de información. El APDU, siempre cuenta con un ASDU de longitud variable.
- **01:** Para tramas de tipo S, *numbered supervisory functions*. Realiza diversas funciones de supervisión, y su APDU consta únicamente de cabecera APCI.
- **11:** Para tramas de tipo U, *unnumbered control functions*. Al igual que las tramas S, su APDU consta únicamente de cabecera APCI. Cuenta con las funciones TESTFR[8], STOPDT[9] y STARTDT[10], las cuales no pueden activarse simultáneamente. Estas funciones cuentan con mecanismos de activación (act) y confirmación (con).

[5] Organización Internacional de Estandarizaciones/Sistema de Interconexiones Abierto.
[6] 64 en hexadecimal. Valor en decimal: 104.
[7] Protocol Data Unit.
[8] La trama es de tipo test.
[9] Parada de transmisión de datos.
[10] Comienzo de transmisión de datos.

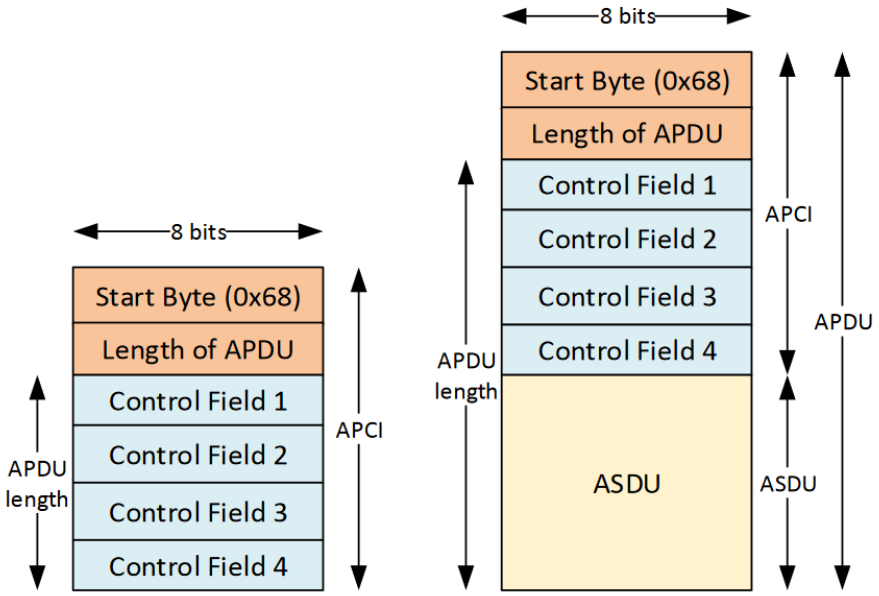


Figura 2.3: Cabecera del protocolo IEC 60870-5-104 [2].

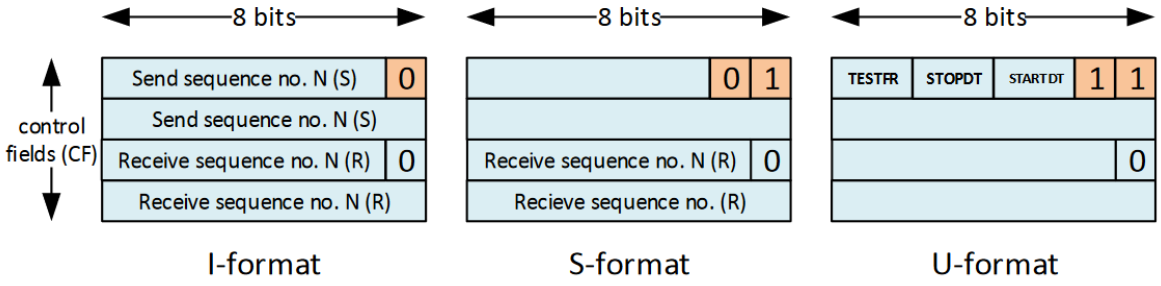


Figura 2.4: Octetos tramas tipo I, S y U [2].

Como hemos dicho anteriormente, los formatos S y U cuentan únicamente con la cabecera APCI, por ello su longitud será fija, y tendrá un valor de 4. Sin embargo, los paquetes con formato I cuentan con la cabecera adicional ASDU, como se ve en la **Figura 2.5**, de tamaño variable.

Esta cabecera está formada por los siguientes campos:

- **Tipo de identificación (*Type identification*), TypeID:** Formada por 1 Byte, según el valor que tome se sabrá la función del paquete. Toma valores de 1-127, el 0 no se usa, de 128-138 reservados para routing y de 136-255 para casos especiales. Se pueden ver las funciones en la **Tabla 2.3**
- **Calificador de estructura (*Structure Qualifier*), SQ:** Formado por un único bit, especifica el método de direccionamiento de los objetos.
- **Número de objetos (*Number of Objets*):** Formado por los 7 bits restantes del octeto, especifica el numero de objetos que tendrá.
- ***Test* (T):** Formado por 1 bit, indica si se encuentra en condición de test.

- ***Positivo/Negativo* (P/N):** Formado por 1 bit, indica la confirmación positiva o negativa de una activación solicitada por una función de aplicación primaria.
- **Causa de la transmission (*Cause of transmission*), COT:** Formado por 6 bits, se utiliza para controlar el enrutamiento de mensajes, dirigiendo el ASDU al programa o tarea correcta para su procesamiento. En la **Tabla 2.3**, se pueden observar los distintos valores que toma.
- ***Originator Address* (ORG):** Formado por 1 Byte, es un campo opcional, sirve para identificar a la estación de control.
- ***ASDU Address Field* (Common Address of ASDU, COA):** Formado por 2 Bytes, está asociada a todos los objetos del ASDU. Se usa normalmente como la dirección de la estación (Broadcast).
- **Dirección de objeto de información (*Information Object Address*), IOA:** Formada por 3 Bytes, indica la dirección destino utilizada por el control de dirección y en el caso de la dirección de monitor, indica la dirección origen.

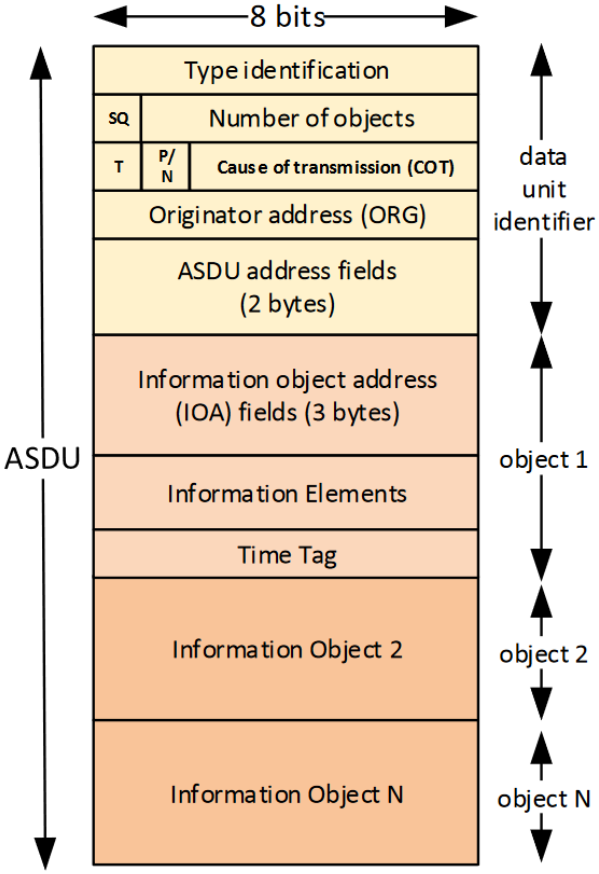


Figura 2.5: Formato ASDU [2].

Type	Description	Reference	Format	Valid COTs
Process information in monitor direction :				
1	Single point information	M_SP_NA_1	SIQ	2,3,5,11,20,20+G
2	Single point information with time tag	M_SP_TA_1	SIQ + CP24Time2a	3,5,11,12
3	Double point information	M_DP_NA_1	DIQ	2,3,5,11,12,20,20+G
4	Double point information with time tag	M_DP_TA_1	DIQ + CP24Time2a	3,5,11,12
5	Step position information	M_ST_NA_1	VTI + QDS	2,3,5,11,12,20,20+G
6	Step position information with time tag	M_ST_TA_1	VTI + QDS + CP24Time2a	2,3,5,11,12
7	Bit string of 32 bit	M_BO_NA_1	BSI + QDS	2,3,5,11,12,20,20+G
8	Bit string of 32 bit with time tag	M_BO_TA_1	BSI + QDS + CP24Time2a	3,5
9	Measured value, normalized value	M_ME_NA_1	NVA + QDS	2,3,5,11,12,20,20+G
10	Measured value, normalized value with time tag	M_ME_TA_1	NVA + QDS + CP24Time2a	3,5
11	Measured value, scaled value	M_ME_NB_1	SVA + QDS	2,3,5,11,12,20,20+G
12	Measured value, scaled value with time tag	M_ME_TB_1	SVA + QDS + CP24Time2a	3,5
13	Measured value, short floating point value	M_ME_NC_1	IEEE STD 754 + QDS	2,3,5,11,12,20,20+G
14	Measured value, short floating point value with time tag	M_ME_TC_1	IEEE STD 754 + QDS + CP24Time2a	2,3,5,11,12,20,20+G
15	Integrated totals	M_IT_NA_1	BCR	2,37,37+G
16	Integrated totals with time tag	M_IT_TA_1	BCR + CP24Time2a	3,37,37+G
17	Event of protection equipment with time tag	M_EP_TA_1	CP16Time2a + CP24Time2a	3
18	Packed start events of protection equipment with time tag	M_EP_TB_1	SEP + QDP +C P16Time2a + CP24Time2a	3
19	Packed output circuit information of protection equipment with time tag	M_EP_TC_1	OCI + QDP + CP16Time2a + CP24Time2a	3
20	Packed single-point information with status change detection	M_PS_NA_1	SCD+QDS	2,3,5,11,12,20,20+G
21	Measured value, normalized value without quality descriptor	M_ME_ND_1	NVA	1,2,3,5,11,12,20,20+G
Process telegrams with long time tag (7 octets) :				
30	Single point information with time tag CP56Time2a	M_SP_TB_1	SIQ + CP56Time2a	3,5,11,12
31	Double point information with time tag CP56Time2a	M_DP_TB_1	DIQ + CP56Time2a	3,5,11,12
32	Step position information with time tag CP56Time2a	M_ST_TB_1	VTI + QDS + CP56Time2a	2,3,5,11,12
33	Bit string of 32 bit with time tag CP56Time2a	M_BO_TB_1	BSI + QDS + CP56Time2a	3,5
34	Measured value, normalized value with time tag CP56Time2a	M_ME_TD_1	NVA + QDS + CP56Time2a	3,5
35	Measured value, scaled value with time tag CP56Time2a	M_ME_TE_1	SVA + QDS + CP56Time2a	3,5
36	Measured value, short floating point value with time tag CP56Time2a	M_ME_TF_1	IEEE STD 754 + QDS + CP56Time2a	2,3,5,11,12,20,20+G
37	Integrated totals with time tag CP56Time2a	M_IT_TB_1	BCR + CP56Time2a	3,37,37+G
38	Event of protection equipment with time tag CP56Time2a	M_EP_TD_1	CP16Time2a + CP56Time2a	3
39	Packed start events of protection equipment with time tag CP56time2a	M_EP_TE_1	SEP + QDP + CP16Time2a + CP56Time2a	3
40	Packed output circuit information of protection equipment with time tag CP56Time2a	M_EP_TF_1	OCI + QDP + CP16Time2a + CP56Time2a	3
Process information in control direction :				
45	Single command	C_SC_NA_1	SCO	6,7,8,9,10,44,45,46,47
46	Double command	C_DC_NA_1	DCO	6,7,8,9,10,44,45,46,47
47	Regulating step command	C_RC_NA_1	RCO	6,7,8,9,10,44,45,46,47
48	Setpoint command, normalized value	C_SE_NA_1	NVA + QOS	6,7,8,9,10,44,45,46,47
49	Setpoint command, scaled value	C_SE_NB_1	SVA + QOS	6,7,8,9,10,44,45,46,47
50	Setpoint command, short floating point value	C_SE_NC_1	IEEE STD 754 + QOS	6,7,8,9,10,44,45,46,47
51	Bit string 32 bit	C_BO_NA_1	BSI	6,7,8,9,10,44,45,46,47
Command telegrams with long time tag (7 octets) :				
58	Single command with time tag CP56Time2a	C_SC_TA_1		
59	Double command with time tag CP56Time2a	C_DC_TA_1		
60	Regulating step command with time tag CP56Time2a	C_RC_TA_1		
61	Setpoint command, normalized value with time tag CP56Time2a	C_SE_TA_1		
62	Setpoint command, scaled value with time tag CP56Time2a	C_SE_TB_1		
63	Setpoint command, short floating point value with time tag CP56Time2a	C_SE_TC_1		
64	Bit string 32 bit with time tag CP56Time2a	C_BO_TA_1		
System information in monitor direction :				
70	End of initialization	M_EI_NA_1	COI	4
System information in control direction				
100	(General-) Interrogation command	C_IC_NA_1	QOI	6,7,8,9,10,44,45,46,47
101	Counter interrogation command	C_CI_NA_1	QCC	6,7,8,9,10,44,45,46,47
102	Read command	C_RD_NA_1	null	5
103	Clock synchronization command	C_CS_NA_1	CP56Time2a	3,6,7,44,45,46,47
104	(IEC 101) Test command	C_TS_NB_1	FBP	6,7,44,45,46,47
105	Reset process command	C_RP_NC_1	QRP	6,7,44,45,46,47
106	(IEC 101) Delay acquisition command	C_CD_NA_1	CP16Time2a	6,7,44,45,46,47
107	Test command with time tag CP56Time2a	C_TS_TA_1		
Parameter in control direction :				
110	Parameter of measured value, normalized value	P_ME_NA_1	NVA + QPM	6,7,9,10,20,20+G,44,45,46,47
111	Parameter of measured value, scaled value	P_ME_NB_1	SVA + QPM	6,7,20,20+G,44,45,46,47
112	Parameter of measured value, short floating point value	P_ME_NC_1	IEEE STD 754 + QPM	6,7,20,20+G,44,45,46,47
113	Parameter activation	P_AC_NA_1	QPA	6,7,8,9,44,45,46,47
File transfer:				
120	File ready	F_FR_NA_1	NOF + LOF + FRQ	13
121	Section ready	F_SR_NA_1	NOF + NOS + LOF + SRQ	13
122	Call directory, select file, call file, call section	F_SC_NA_1	NOF + NOS + SCQ	5,13
123	Last section, last segment	F_LS_NA_1	NOF + NOS + LSQ + CHS	13
124	Ack file, Ack section	F_AF_NA_1	NOF + NOS + AFQ	13
125	Segment	F_SG_NA_1	NOF + NOS + LOS + segment	13
126	Directory	F_DR_TA_1	NOF + LOF + SOF + CP56Time2a	3,5
127	QueryLog - Request archive file	F_SC_NB_1		

Tabla 2.2: Tipos de Type ID y su descripción ASDU para IEC104 [2].

2.4. Trabajos previos

Este Trabajo Fin de Grado, como hemos mencionado con anterioridad, se basa en otro previo realizado por David Abreu Cañamares[5], cuyos scripts se encuentran alojados en un repositorio de GitHub[9].

La finalidad de dicho trabajo era la misma, sin embargo, el desarrollo es distinto. Mientras que el presente trabajo está desarrollado en C, el trabajo de David Abreu Cañamares se desarrolló en Python. El hecho de realizarlo en Python, usando la librería *SCAPY*, implicó que los tiempos de ejecución no fueran los esperados, superando incluso el tiempo que duraba la captura de los paquetes en la red. Por este motivo, se decidió volver a plantear el trabajo, pero esta vez con desarrollo en C, junto a la librería libpcap de la que hablaremos en el capítulo 4, debido a su mayor velocidad en cuanto a tiempos de ejecución y tiempos de energía, como podemos ver en la publicación de *The New Stack*[6].

Para la generación de ataques, sirvió de ayuda un repositorio de GitHub[4], que contaba con un simulador de cliente y servidor usando el protocolo IEC 60870-5-104 tratado en este trabajo. La modificación de los parámetros de dichos paquetes enviados, sirvió para poder realizar pruebas de validación del detector de anomalías.

2.5. Detección de anomalías en el tráfico de red

En este Trabajo Fin de Grado se realizará un sistema IDS (*Intrusion Detection System*) cuyo cometido principal es detectar ataques y notificarlos. Este sistema no será capaz de detener un ataque, pero sí informar al usuario de las anomalías que presente la red en tiempo real. Se encargará de monitorizar la red, comparándolo con una linea de base establecida. La línea base determina lo que se considera normal para la red en términos de ancho de banda, protocolos, puertos y otros dispositivos, y el IDS alerta al administrador de todo tipo de actividad inusual.

Uno de los sistemas IDS más usados en la actualidad es *Snort*[11]. Es un sistema de detección de intrusos en red, libre y gratuito creado por *Sourcefire*, empresa posteriormente adquirida por *Cisco Systems*. Este programa, se encarga de detectar cuando un paquete de red no coincide con algún patrón establecido en las reglas. Para el caso del protocolo IEC 60870-5-104, las reglas establecidas únicamente alertan al usuario ante ciertos *TypeID* o cuando es desconocido y cuando el tráfico de la red proviene de una red externa[10]. El sistema IDS desarrollado en este trabajo fin de grado amplía esta detección de anomalías, como se puede ver en el capítulo 4.

Tanto el programa detector realizado en este trabajo como el *Snort* presentan la siguiente estructura: (ver **Figura 2.6.**)

[11] `https://www.snort.org/`

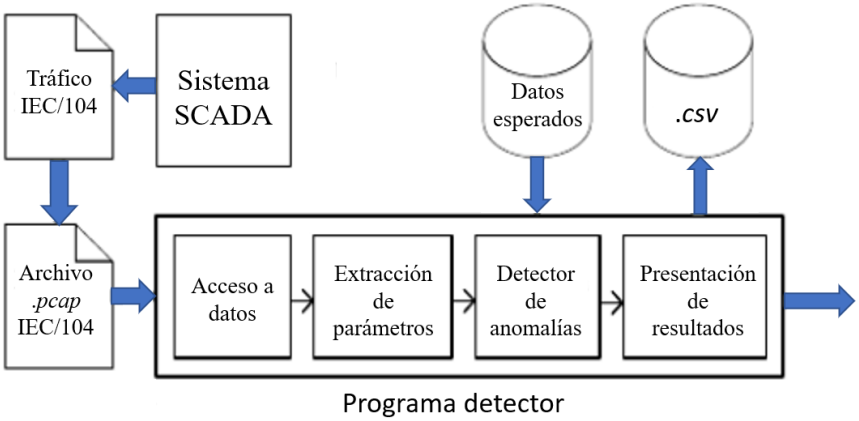


Figura 2.6: Proceso llevado a cabo durante la ejecución del programa. [3]

2.6. Conclusión

En este capítulo, han quedado definidas las bases de las redes SCADA, así como, la estructura del protocolo IEC 60870-5-104 , introduciendo los términos ASDU, APCI y APDU, junto a todos los campos pertenecientes a esta estructura que han servido para establecer las medidas de detección en el programa (ver Capítulo 4.). También introducimos la **Tabla 2.3**, que se usará para el desarrollo del sistema detector, en concreto se dará especial importancia a los valores del *TypeID* y de la *COT* correspondientes a cada valor. Además, se ha hecho mención al trabajo previo del que parte este desarrollo, introduciendo el objetivo a batir de este proyecto.

Por último, se ha comentado el método de detección que se ha llevado a cabo, incluyendo un ejemplo de programa con un cometido similar.

3

Análisis de Requisitos

3.1. Introducción

En este capítulo, una vez conocidas todas las características del protocolo tratado y naturaleza de las redes SCADA, nos centramos en explicar los requisitos que debemos cumplir, para dar como satisfactorio el resultado del detector de anomalías.

Los dividiremos en requisitos funcionales, aquellas funciones o comportamientos con las que debe contar el sistema, y requisitos no funcionales, que son los criterios que permitirán evaluar la funcionalidad del sistema.

3.2. Requisitos funcionales

El sistema eficiente de análisis del protocolo IEC 60870-5-104 para la detección de anomalías, cuenta con una serie de requisitos que se espera que se lleven a cabo para su correcta implementación.

El primero de ellos, es la correcta obtención de los campos del tráfico de red capturados, requeridos para su posterior análisis. Un fallo en esta función, y el cometido del detector se verá truncado, generando posibles alertas falsas. También se exige que esta extracción de datos se realice de manera óptima.

Otro requisito fundamental para el desarrollo del detector, es la correcta implementación de mecanismos de alarma que analicen los datos obtenidos, y avisen al usuario si se está produciendo alguna anomalía. También, se requiere que dicho análisis se pueda exportar, para así poder realizar un posterior estudio de dichas anomalías y establecer patrones si así se desea.

3.3. Requisitos no funcionales

En cuanto a requisitos no funcionales el más importante, y el motivo para la realización de este Trabajo Fin de Grado, es el rendimiento. De este sistema implementado en C, se espera que los tiempos de ejecución, con respecto al trabajo homólogo implementado en Python, posean varias ordenes de magnitud menor. La inmediatez en el análisis de anomalías en estos sistemas, es fundamental para poder garantizar la seguridad en los mismos. Es por ello, que el objetivo primordial de este trabajo es lograr dicha finalidad.

Otro requisito de gran importancia es la fiabilidad. Se espera que el detector obtenga y analice los datos de manera precisa.

También, se requiere la realización de pruebas para verificar el correcto funcionamiento del sistema detector. Para ello, se generarán anomalías controladas, de manera que se pueda verificar que se está detectando adecuadamente.

3.4. Conclusión

Con este capitulo, se han especificado los requisitos básicos en los que se ha basado el desarrollo y posterior validación del detector de anomalías, diferenciándolos en funcionales y no funcionales. Estos, servirán como medio de verificación para establecer si se han alcanzado los objetivos deseados y posibles modificaciones para un trabajo futuro.

4

Diseño y Desarrollo

4.1. Introducción

En este capitulo, desarrollaremos las fases llevadas a cabo para la realización del programa de análisis y detección de anomalías en el entorno de redes SCADA. Se comenzará explicando el entorno usado para la realización del detector y las posteriores verificaciones. Se continuará describiendo la arquitectura desarrollada, comenzando con la fase del análisis de tráfico, posteriormente la comprobación de anomalías y finalizando con la presentación de los resultados. Por último, se comentarán las mejoras llevadas a cabo en el detector.

4.2. Entorno de desarrollo

Para la realización de este trabajo se ha optado por utilizar una máquina virtual, usada para desarrollar el detector eficiente. Más adelante, durante la fase de validación, se requirió una segunda máquina virtual.

Se ha decidido usar VMware debido a la facilidad que supone el crear nuevos sistemas con Ubuntu, la distribución de Linux elegida, para así poder generar el envío de paquetes durante el proceso de validación. Para que ambas máquinas puedan conectarse entre ellas y poder transmitirse datos, es necesario que cuenten con una configuración concreta. En la pestaña de configuración, dentro de VMware, en la subsección *Network Adapter* es necesario que se encuentre en modo NAT o en Host-only. Con ello, aplicado a ambas maquinas virtuales, se consigue que ambas se encuentren en la misma subred. De esta manera, se establecerá en cada máquina virtual una función distinta, una hará la función de maestro y otra de esclavo.

También es necesario añadir que, para poder compilar el programa, se ha necesitado instalar la librería libpcap. Se ha usado el siguiente comando:

```
apt-get install libpcap0.8-dev
```

4.3. Arquitectura desarrollada

En este detector se prima la velocidad de ejecución, que es el objetivo principal del proyecto. Es por ello, que se modificará ciertos parámetros a lo largo del desarrollo con el fin de optimizar estos tiempos. La idea primordial es conseguir reducir los tiempos de ejecución por debajo de los logrados en trabajos anteriores. También añadir, que se desea que estos lleguen a ser inferiores al tiempo que dura el tráfico a analizar.

4.3.1. Análisis de tráfico

Como se ha comentado previamente en este trabajo, el lenguaje de programación usado para el desarrollo del detector es C, pudiendo gracias al mismo reducir de forma drástica los tiempo de ejecución respeto al trabajo previo hecho en Python. El motivo de la lentitud viene de parte de la librería Scapy la cual cuenta con altos tiempos de ejecución.

Es necesario también comentar, que durante el desarrollo del sistema detector, se ha seguido un diseño por filtros, por lo que si algún campo del protocolo IEC 60870-6-104 no cumple con las características esperadas se considerará anomalía, no permitiendo una segunda valoración.

Para comenzar el análisis del tráfico, el programa comienza leyendo un archivo *.pcap* con la función `pcap_open_offline(const char *fname, char *errbuf)`, pasando como argumentos el nombre del archivo a leer y una variable para volcar los posibles errores, respectivamente. Con la función `pcap_next_ex` se recorre el archivo, paquete a paquete, devolviendo 1 si se ha leído un paquete o -2 si la traza ha terminado.[11]

Se inicia entonces la obtención de los datos de cada uno de los paquetes. Para ello, se recorre cada paquete extrayendo los datos de interés haciendo uso de estructuras, donde se establece el tamaño de cada campo, como se ve en la **Figura 4.1**. En las primeras fases de desarrollo se optó primero por extraer la totalidad de los datos presentados por el paquete, siendo más adelante cuando se decide cuales usar y cuales no, de cara a permitir el análisis de anomalías.

Los paquetes se analizan dato a dato, primero tras la extracción de la cabecera Ethernet se realiza una comprobación para establecer si el paquete es transportado por TCP, parando el análisis de dicho paquete si este protocolo no se encuentra presente. Un aspecto muy importante para el análisis es la extracción del puerto origen y destino, el cual, según el protocolo tratado en el trabajo, establece que el puerto origen del esclavo ha de ser el 2404, y por lo tanto, el puerto destino del maestro ha de ser el 2404.

Cabe destacar, el cálculo de la variable *TCP Length*, la cual índica el tamaño de los datos en un paquete TCP. Para ello, haremos uso de una variable extraída de la cabecera IP, conocida como *Total Length*, la cual indica el tamaño total del paquete en bytes. A esta variable, se le resta la suma del tamaño de la cabecera IP más el campo *Data Offset*,

```c
/* IEC 60870-5-104 header */
struct sniff_104{
    uint8_t start;              /* START */
    uint8_t apdu_length;        /* Length of APDU headerm */
    uint8_t octet_1;            /* Octets for FORMAT */
    uint8_t octet_2;            /* Octets for FORMAT */
    uint8_t octet_3;            /* Octets for FORMAT */
    uint8_t octet_4;            /* Octets for FORMAT */
    uint8_t type;               /* Type identification */
    uint8_t sq_numIx;           /* Structure qualifier 1 bit */ /* Number of objects */
    uint8_t causeTx;            /* Cause of transmission */
                                /* Test 1 bit */
                                /* Positive or Negative 1 bit*/
    uint8_t oa;                 /* Originator address */
    uint16_t addr;              /* ASDU addres fields */
};

struct ioa{
    uint32_t ioa;               /* Information objects address fields */
};
```

Figura 4.1: Ejemplo de estructura usada para la lectura de datos.

el cual indica el tamaño de la cabecera TCP, todo multiplicado por 4, ya que ambas están formadas por palabras de 32 bits. Es decir:

```
TCP Length=(Total Length)-(IP HeatherLength + Data Offset)*4
```

Una vez conocidos los puertos, si uno de ellos es el 2404, y el campo *TCP Length* calculado es distinto de 0, se empieza la lectura del segmento con protocolo IEC 60870-5-104. En ella se extraen los datos necesarios, que posteriormente comentaremos, para el análisis de anomalías. Se establece una distinción de los datos extraídos según el formato con el que cuente el paquete leído. Hablamos de los tipos de formato: U, S, I.

Aquellos paquetes con formato I, cuentan con una implementación de tablas *hash*, donde se almacenan la dirección IP origen, el *TypeID* y el número de veces que aparece esa combinación, datos que podrán ser útiles para un análisis posterior. También se crea una segunda tabla *hash*, donde se almacenan las direcciones IP origen y destino de los paquetes considerados como esclavos. La estructura de estas tablas *hash* se puede ver en la **Figura 4.2**, con los datos almacenados en ellas plasmados en las **Figuras 4.3 y 4.4**.

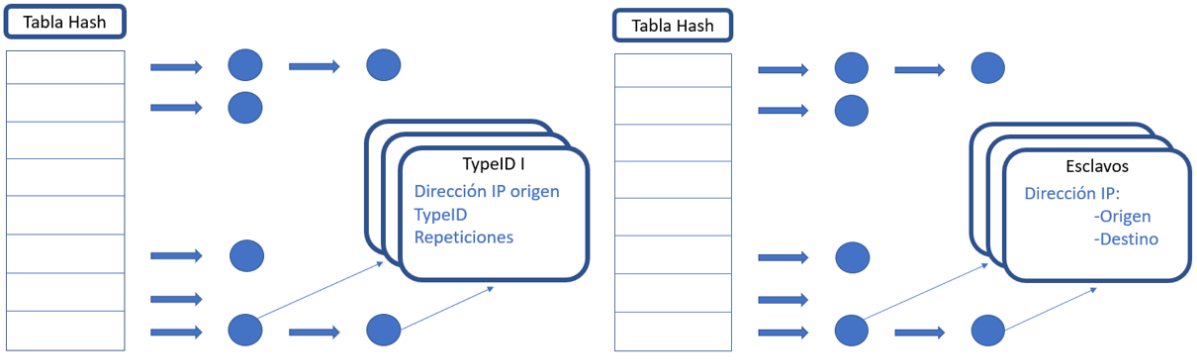


Figura 4.2: Estructura tabla *hash*.

Por último, cabe destacar la implementación de listas, en las cuales se incluyen las direcciones IP origen y destino junto al número de veces que aparece dicha combinación en aquellos paquetes que están malformados.

```
typedef struct node {
    uint8_t ipsrc[IP_ALEN];
    uint8_t ipdst[IP_ALEN];
    struct node* next;
}node;
```

Figura 4.3: Estructura guardadas en la tabla *hash* de esclavos.

```
typedef struct DataItem {
    uint8_t ipsrc[IP_ALEN];
    uint8_t type;
    uint8_t counter;
    struct DataItem* next;
}DataItem;
```

Figura 4.4: Estructura guardadas en la tabla *hash* de paquetes tipo I.

4.3.2. Comprobación de anomalías

Una vez conocidos los campos del paquete, se prosigue a realizar las comprobaciones pertinentes que permitan determinar si hay o no hay anomalías.

Se comienza comprobando si el paquete es transportado por TCP y como hemos dicho previamente paramos la lectura de ese paquete en caso contrario. También mostramos un mensaje por pantalla avisando de esta anomalía, indicando el número de paquete, para así permitir un segundo análisis si así se desea.

Se continúa comprobando el número del puerto origen y destino, que como hemos dicho en el apartado anterior, uno de los dos ha de estar conectado al puerto genérico 2404. Si no es el caso, se muestra por pantalla un mensaje, indicando la anomalía y el número del paquete en el que ocurre, al igual que en la primera comprobación. Se considera una anomalía, debido a que en este tipo de redes no debería verse tráfico de otro tipo, como se explicó en el Trabajo Fin de Grado de Álvaro Culebras[12].

Se prosigue analizando los diferentes campos del protocolo, centrándonos ahora en los formatos PDU. Una vez conocido a qué formato pertenece se comprueba el campo *ApduLen*, el cual debe de ser 4 en los paquetes con formato U y S, y mayor que 4 en los I. Si no ocurre, nuevamente se muestra por pantalla como en los casos anteriores.

Para los paquetes del tipo U, se establecerá como anomalía cuando ninguna de las funciones de dicho formato (TESTFR, STOPDT y STARTDT) es activada. Cuando esto ocurra se notificará por pantalla como anomalía.

Después, nos centraremos en los paquetes que contienen datos, los del formato I. La primera comprobación a realizar es la detección de paquetes malformados. Para ello, se comprueba que la suma de todo el tamaño de los objetos presentes en el paquete es igual al tamaño del *ApduLen* de ese paquete. Si no ocurre, se establecerá que el paquete está malformado y se alertará de la anomalía.

A continuación se prosigue con la obtención del *TypeID*, el cual según la estructura del protocolo, el valor nunca puede ser el 0 y los valores por encima de 127 son exclusivos

para casos concretos. Por tanto, si el *TypeID* no se encuentra entre el 1 y el 127 se vuelve a alertar de la anomalía.

La comprobación de anomalías continúa fijándonos en el valor del puerto y el número de *TypeID* en su conjunto. En esta comprobación, se evalúa si el paquete realiza adecuadamente su función de maestro, es decir, el envío de órdenes e información, o de esclavo, que son respuestas al maestro o envío de información.

- Los paquetes considerados como enviados por un maestro son aquellos que tienen como puerto destino el 2404 y uno de los siguientes *TypeID*:

  {45-51, 58-64, 100-105, 107, 110-113}

- Mientras que los paquetes considerados como enviados por un esclavo son aquellos que tienen como puerto origen el 2404 y uno de los siguientes *TypeID*:

  {1, 3, 5, 7, 9, 11, 13, 15, 20-21, 30-40, 45-51,
  58-64, 70, 100-101, 103-105, 107, 110-113}

De nuevo, si no cumple con lo anterior se muestra por pantalla la anomalía y se informa del número del paquete, al igual que en el resto de casos.

El siguiente elemento sometido a análisis es el *structure qualifier*, el cual posee la información acerca del número de objetos de información. Éste tiene un máximo de 127 objetos, por lo que si lo supera se avisa de la anomalía.

Continuamos con las anomalías, esta vez centradas en la *COT* o *Cause of Transmission*. Como se ha visto en la **Tabla 2.3**, cada valor del *TypeID* puede tener ciertos valores de la *COT*. Por tanto, al igual que con el resto de casos, si no cumple uno de esos valores, se muestra por pantalla la anomalía.

Para el caso del *TypeID*=45-48,100,101 [3], el valor de *COT* deberá ser 6 en la dirección de control, y 6 o 7 en la dirección de monitor. Por tanto, cualquier valor que no sean estos se notificarán como anomalía.

4.3.3. Presentación de resultados

Tras la extracción de todos los datos de la traza analizada, así como sus anomalías, se prosigue con la representación de los resultados. Durante el análisis, se ha llevado a cabo una categorización del número de paquetes totales, número de paquetes que cumplen el protocolo IEC 60870-5-104, número de paquetes de cada tipo (S, I, U), número de anomalías, las IP origen y destino de los paquetes malformados y la IP origen junto al *TypeID* de los paquetes con formato I.

También cuenta con la posibilidad de guardar los datos de los paquetes malformados en un archivo con extensión *.csv*, permitiendo un posterior análisis con cuadros estadísticos si así se deseara.

Por último, el sistema calcula el tiempo que tardó el detector en analizar la traza, mirando el tiempo al inicio y al final del análisis. También se realiza un cálculo del *throughput*. Este, se calcula dividiendo el numero de bytes totales analizados entre el tiempo de ejecución calculado tras el análisis completo. Estos datos dan información relevante acerca del rendimiento del programa.

4.4. Mejoras

En esta sección nos centraremos en comentar las diferentes mejoras que se han llevado a cabo a lo largo del desarrollo del sistema detector eficiente.

Una vez desarrollado el sistema detector se prosiguió a su optimización. Para ello, se usó una herramienta de Valgrind, en concreto *Callgrind*, la cual permitió analizar el código y extraer los tiempos de ejecución de las diferentes funciones de dicho código.

Para ello, previa instalación de la herramienta se ejecuta el siguiente comando en el terminal: `valgrind --tool=callgrind ./iec104 -f - traza.pcap`

- Con ello se ejecutará el código del detector con los resultados pertinentes, pero a mayores, al final de la ejecución, se mostrarán unos mensajes como los de la **Figura 4.5**. Nos interesa el primer valor que se encuentra entre == ==. Ese será el número de ejecución que usaremos en el siguiente comando: `sudo kcachegrind callgrind.out.23770`

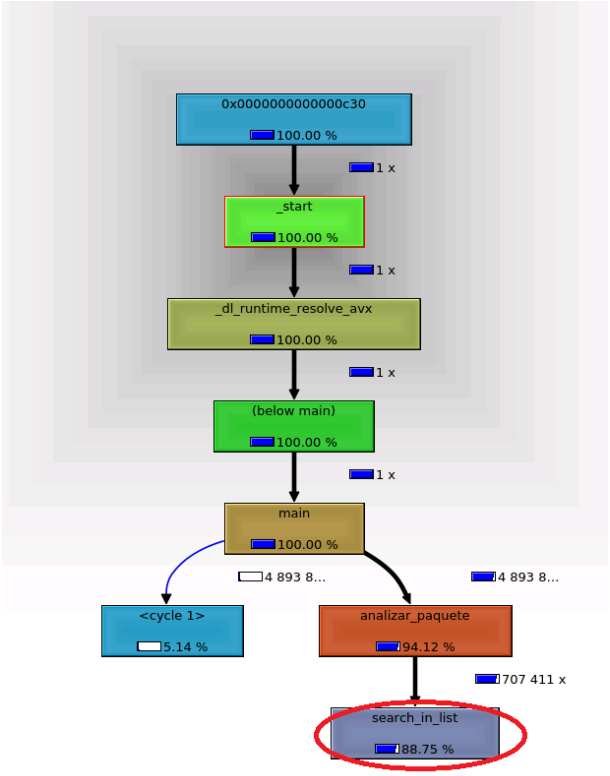
```
==23770==
==23770== Events    : Ir
==23770== Collected : 4693856069
==23770==
==23770== I   refs:      4,693,856,069
```

Figura 4.5: Resultados ejecución `valgrind --tool=callgrind`.

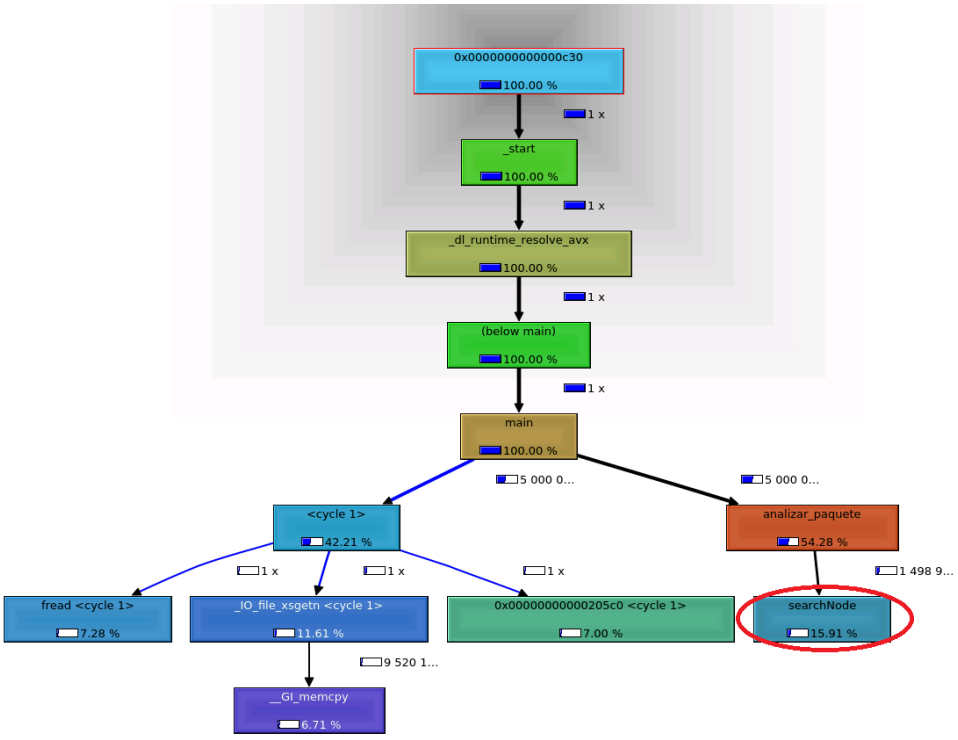
Aplicada a la ejecución del código desarrollado, se pudo apreciar cómo en la búsqueda de datos de la lista enlazada, donde se almacenaban la dirección IP origen, el *TypeID* y el número de veces que aparecía esa combinación, ocupaba aproximadamente el 89 % del tiempo total de ejecución.

Se decidió cambiar dicha lista enlazada por una tabla *hash* con detector de colisiones, la cual permitiría un acceso a los datos mucho más rápido, debido a su asociación de los datos con una clave concreta.

Como se puede apreciar en la **Figura 4.6 (a)**, la función *search in list* era el 88.75 % del total de la ejecución, dando claras señales de que la lista enlazada no era adecuada para esta implantación. Mientras que en la **Figura 4.6 (b)** con implantación de tablas *hash* la función para buscar en la tabla ocupa un porcentaje mucho menor del tiempo de ejecución. Debido a que la implementación de estas tablas cuentan con detección de colisiones, aparece un 15.91 % asociado a las listas enlazadas, que poseen aquellos datos almacenados cuya clave (key) coincide.



(a) Listas enlazadas, 88.75 %.



(b) Tabla *hash*, 15.91 %.

Figura 4.6: Porcentaje de tiempo requerido por cada instrucción.

4.5. Conclusión

Fijadas y detalladas todas las acciones llevadas a cabo por el programa detector, se da por finalizado el desarrollo del detector de anomalías del protocolo IEC 60870-5-104, pasando a la etapa de validación, donde se probará su funcionamiento y su velocidad de ejecución.

Se han obtenido los datos de los paquetes capturados, haciendo uso de estructuras de manera que quedaban prefijados los tamaños de cada campo, como se ha podido apreciar en la **Figura 4.1**. Posteriormente estos datos se han contrastado con los valores esperados según las reglas, avisando de anomalías en el caso de que las haya.

A través de este desarrollo se han podido comprobar las claras ventajas de las tablas *hash* frente a las listas, por lo que se decidió sustituirlas para así obtener mejores tiempos de ejecución.

5 Validación

5.1. Introducción

En este capitulo nos centraremos en el análisis de la funcionalidad del sistema detector desarrollado en un escenario con datos reales.

Detallaremos todas las pruebas realizadas, tanto los ataques provocados, como el análisis de un ataque real facilitado a través de la Cátedra UAM-Naudit por medio de una captura de tráfico red, con extensión *.pcap*. También, se comentará los métodos de verificación usados y las anomalías detectadas en dichos métodos.

5.2. Tráfico legítimo

Una de las primeras pruebas que se usaron como validación era un archivo *.pcap*, suministrado por un usuario de *GitHub* [13], que contaba con el protocolo IEC 60870-5-104.

Con este archivo, se pudo verificar al comienzo del desarrollo del programa detector, que se estaban extrayendo adecuadamente todo los datos necesarios. Para esto también se usó Wireshark[1], un analizador de protocolos del que hablaremos más adelante, en el cual se podían ver todos los datos ya extraídos de la traza.

Como se puede ver en la **Tabla 5.1**, los tiempo de ejecución son muy bajos, pero debido a que esta traza no contaba con ningún tipo de anomalía y su tamaño era muy reducido, era necesario proseguir la validación con archivos de mayor volumen.

[1] `https://www.wireshark.org/`

Se procesaron 105 paquetes.
Ataques: 0
Los cuales el 100 % de ellos fueron TCP: 105
Los cuales el 0.000000 % de ellos fueron UDP: 0
Los cuales el 60.952381 % de ellos fueron IEC104: 64

IP	Tipo	Cantidad	Porcentaje
10.20.102.1	[45]	[2]	2.197802 %
10.20.100.108	[30]	[3]	3.296703 %
...	...	...	...

Los cuales el 86.666667 % de ellos fueron paquetes I: 91
Los cuales el 13.333333 % de ellos fueron paquetes S: 14
Los cuales el 9.523810 % de ellos fueron paquetes U: 10
TESTFR act: 4
TESTFR con: 4
STARTDT act: 1
STARTDT con: 1
STOPDT act: 0
STOPDT con: 0

Ataques: 0

Malformed:

IP origen	IP destino	Cantidad

Maestro erróneo: 0
El programa ha tardado 0.000852 segundos en ejecutar.
El throughput es de 9815727 Bytes/segundo.

Tabla 5.1: Resultados tras la ejecución de la traza de GitHub. Se ha obviado la lista de IP y tipo por simplificación.

5.3. Tráfico anómalo

Una de las pruebas realizadas para verificar el correcto funcionamiento del detector de anomalías se basa en el código, realizado en Python, proporcionado por el repositorio de GitHub[4], el cual contaba con un simulador de cliente y servidor. Basándose en dicho código, la modificación de parámetros permite verificar el funcionamiento adecuado ante las anomalías, al igual que se hizo en el trabajo de David Abreu Cañamares [5].

Para la generación de paquetes con un *ApduLen*[2] erróneo, se ha de modificar la parte del código donde se le adjudica ese valor. Como se ve en la **Figura 5.1**, se modifica el valor de manera que se genere un dato erróneo

También se usó este método para generar valores de *COT* erróneos. Para ello se modifica la parte del código de cada tipo de transferencia de datos como se ve en la **Figura 5.2**

[2]Longitud de la cabecera APDU.

```
# IEC104 apci
class i_frame(Packet):
    name = "i_frame"
    fields_desc = [ XByteField("START", 0x68),
                    XByteField("ApduLen", None),
                    LEShortField("Tx", 0x0000),
                    LEShortField("Rx", 0x0000),
                    ]

    def post_build(self, p, pay):
        if self.ApduLen is None:
            l = len(pay)+4
            p = p[:1] + struct.pack("!B", l) + p[2:]
        return p+pay+str(3)
```

Figura 5.1: Código transmisión de paquetes con modificación de longitud de APDU[4].

```
class asdu_head(Packet):
    name = "asdu_head"
    fields_desc = [ XByteField("TypeID", 0x45),
                    XByteField("SQ", 0x01),
                    XByteField("Cause", 0x08),  #cambiado el 6 por 8
                    XByteField("OA", 0x04),
                    LEShortField("Addr", 0x0003)]
```

Figura 5.2: Código transmisión de paquetes con modificación de *COT*[4].

Para la comprobación del funcionamiento de la tabla *hash*, que verificaba si un esclavo tenía más de un maestro, se realizó una modificación del programa, de manera que se cambiase la dirección IP de la máquina virtual durante el envío de los paquetes. Debido a que el servidor (esclavo) era siempre el mismo, todos los paquetes enviados por los maestros iban a la misma dirección destino. Esto provoca, que nada más cambiar la dirección IP, el detector ya avise de la anomalía.

Con este fin, se realizó un bucle en el que se cambiase la dirección IP mediante los siguientes comandos, siguiendo ese orden:

```
os.system('sudo ifconfig ens33 down')
os.system('sudo ifconfig ens33 192.168.145.'+str(ip_consecutive))
o.system('sudo ifconfig ens33 up')
```

Siendo *ens33* la interfaz de red de la máquina virtual, por lo que variará según el terminal que se use, al igual que la dirección IP. Se utiliza *str(ip_ consecutive)*, de manera que permita variar a lo largo del bucle el valor, generando por lo tanto, diferentes maestros. El comando *os.system* es la manera que usa Python para realizar cambios en el sistema.

Cabe mencionar que durante el desarrollo de las pruebas, se detectaron fallos inesperados ante la ejecución de los comandos mencionados previamente. Una vez cambiada la dirección IP, cuando se hacía `sudo ifconfig ens33 up` se generaba una ruta duplicada, como se pudo ver a través del comando `route -n`. Esto provocaba que la IP del esclavo estuviera fuera de alcance y por tanto no se establecía intercambio de paquetes.

Para buscar la causa de este error, se siguieron varias medidas. Lo primero que se hizo, cuando se detectó que no se intercambiaban paquetes, es verificar que la dirección IP se cambiaba adecuadamente, haciendo uso del comanto `watch -n 1 ifconfig ens33`. Con ello, se podía ir viendo en tiempo real los valores que iba tomando la dirección IP de la máquina virtual. Gracias a este, se detectó que mediante el comando `sudo`

`ifconfig ens33 192.168.145.'+str(ip_consecutive)` la máquina virtual ya poseía una dirección IP activa. Ante ello, se decidió eliminar el comando `sudo ifconfig ens33 up` del código del programa, solucionando los errores comentados en ese momento.

No obstante, en otra ejecución posterior, se requirió volver a poner el comando eliminado para que funcionase correctamente el cambio de dirección IP. Se dedujo, por tanto, que el error debió de ser ocasionado por la configuración del ordenador.

5.4. Validación a partir de resultados de Wireshark

El uso del programa Wireshark, que es un analizador de protocolos que permite ver todo el tráfico que pasa a través de una red, será de vital importancia a la hora de validar el detector. Su uso fue el examinar los datos proporcionados, a través de un captura de red con formato *.pcap* y compararlo con los datos extraídos a través de detector realizado en este trabajo.

En las primeras etapas del desarrollo del detector, como se ha mencionado previamente, se extrajeron los mismos datos que proporcionaba Wireshark, de manera que se pudo comprobar que se realizaba correctamente dicha extracción.

También sirvió como herramienta de detección de errores. Una vez implantado los avisos de anomalías, se pudieron comprobar ciertos falsos positivos, que fueron subsanados gracias a la comprobación de datos con los proporcionados con Wireshark. Algunos de ellos serán comentados posteriormente en la sección de Tráfico real 5.5.

A su vez, se detectaron ciertas anomalías en el programa en cuestión. Ciertos paquetes, considerados como *TCP retransmission* por Wireshark, la cabecera perteneciente a los datos del protocolo IEC 60870-5-104 no es analizada, como se puede ver en la **Figura 5.3**. Se pudo apreciar, que esto ocurría en aquellos paquetes en los que el RTO[3] es calculado.

También se detectó, en la traza facilitada por Cátedra UAM-Naudit que posteriormente comentaremos, que ciertos paquetes detectados por Wireshark como *TCP Out of Order*, la cabecera del protocolo tratado en este trabajo no era analizada. Como se aprecia **Figura 5.4**, una vez terminada la extracción de los datos de la cabecera TCP los datos son obviados.

5.4.1. Tshark

Con la herramienta Tshark, que es una versión sin interfaz gráfica de Wireshark, se permite verificar que estamos obteniendo los datos adecuadamente.

Se usó sobre la traza real facilitada por la Cátedra UAM-Naudit. Se decidió, debido al gran tamaño de los datos, que una buena manera de comprobar el funcionamiento adecuado del código detector era generando dos archivos con extensión *.csv*. Uno con los paquetes de cada formato (I, U, S) a partir del resultado extraído por el detector realizado por el alumno, y otro archivo extraído del propio *.pcap* con la herramienta Tshark.

[3]Retransmission timeout.

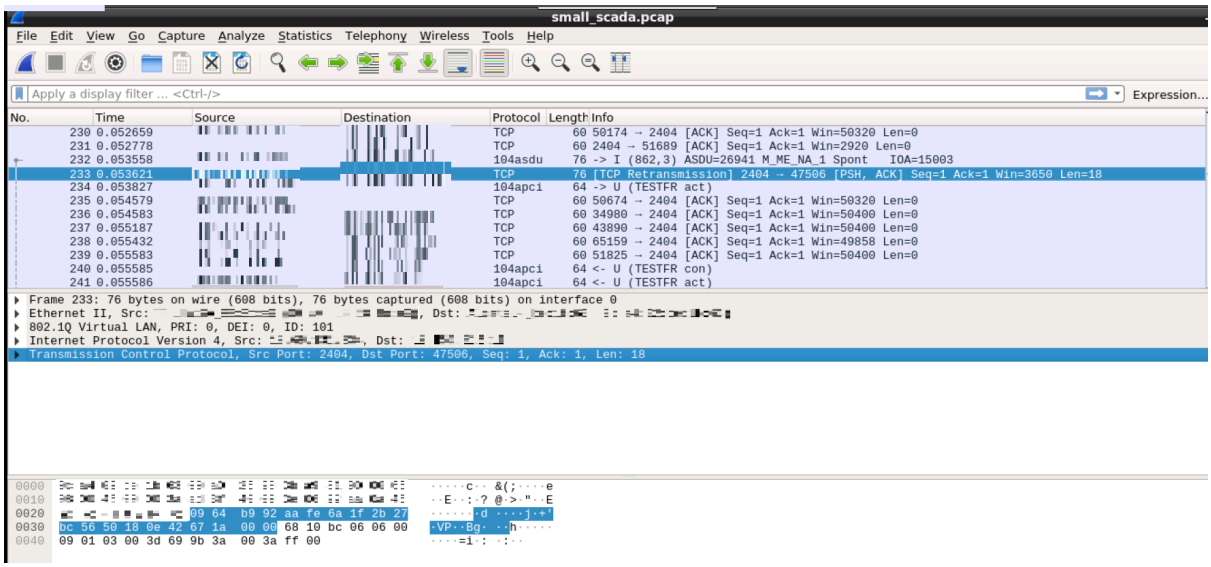


Figura 5.3: Ejemplo de TCP retransmission.

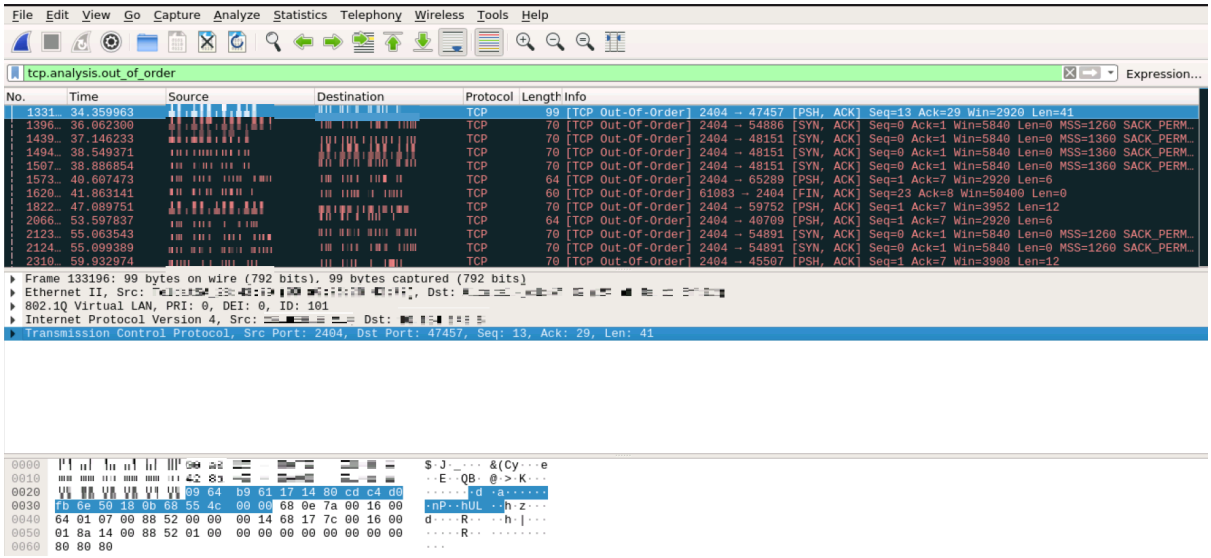


Figura 5.4: Ejemplo de TCP Out of Order.

Para obtener dichos datos con la herramienta se usó los siguientes comandos en la terminal:

- `tshark -r small_scada.pcap -Y ”104apci.type eq 3” -T fields -e frame.number >prueba_u.csv`
- `tshark -r small_scada.pcap -Y ”104apci.type eq 1” -T fields -e frame.number >prueba_s.csv`
- `tshark -r small_scada.pcap -Y ”104apci.type eq 0” -T fields -e frame.number >prueba_i.csv`

Con ello obtendremos el número del paquete que cuenten con ese formato. Para el formato U sería con *104apci.type eq 3*, para el S *104apci.type eq 1* y para el I *104apci.type*

eq 0. Tras ello, se hizo una modificación en el programa detector, de manera que cada vez que se detectara el formato que se quisiera comprobar en ese momento, lo guardase en un archivo *.csv* junto con el número del paquete, al igual que con el Tshark.

Una vez obtenidos ambos archivos, con la herramienta LibreOffice hacemos una comprobación, exportando ambos archivos a un solo documento y comprobando que el número de paquete mostrado coincide en ambos casos. Para esto, se usó la siguiente ecuación:

```
=If(Ax=Bx;"";"no match")
```

Sin embargo, el método de extracción de datos anterior no era suficiente. Los paquetes en los que hubiera varios tipos de formato, es decir aquellos paquetes en los que los datos fueran de tipo I junto a otro tipo distinto, por ejemplo U, S u otro I, no eran extraídos adecuadamente. Se necesitó usar *AWK*. *AWK* es un lenguaje de programación que permite procesar datos basados en texto. Primero se generó un archivo *.txt* que contase con aquellos paquetes en los que hubiera varios tipos de formato juntos. Se usó el siguiente código:

```
tshark -r captura104.pcap -Y "104apci.type eq 0T fields -e frame.number
-e 104apci.type | grep , |less >archivo.txt
```

A continuación, se prosigue a la obtención de los paquetes que contaran con el tipo I. Con el siguiente código se exportaba a un *.csv* todos los números de paquetes en los que había al menos un I junto a un U, S u otro I.

```
awk 'split($2,tipo,",");for (i in tipo) print $1,tipo[i]' archivo.txt
>comprobación.csv
```

Con estas verificaciones se pudo comprobar manualmente cual es el número de paquete en el que el Tshark y el programa implementado, no coincidían. Fue gracias a estos métodos, que el alumno se dio cuenta de las anomalías de Wireshark presentadas en el apartado anterior y mostradas en las **Figuras 5.3 y 5.4**

5.5. Tráfico real

En este apartado hablaremos de los resultado obtenidos analizando la traza facilitada por la Cátedra UAM-Naudit.

Esta traza tiene una duración, de captura de tráfico de red, de 20 minutos y 56 segundos, su tamaño es de 481 MegaBytes y cuenta con 5.000.000 de paquetes. No obstante, siguiendo las consideraciones comentadas en el apartado de Wireshark se decidió eliminar aquellos paquetes que fueran *TCP retransmission* y *TCP Out of Order*, para evitar diferencias entre los resultados que se obtienen por el programa implementado y por Tshark. Eliminados estos paquetes, se prosiguió a realizar las comprobaciones comentadas en el apartado anterior.

Una vez comprobados que los paquetes son extraídos adecuadamente, se prosigue a ejecutar la traza en su totalidad. En la **Tabla 5.2** se muestra los resultados más relevantes, pero para más detalle se puede ver el **Anexo A.1**.

Como se puede apreciar con los resultados, el detector identificó 67 casos anómalos. Estos ataques vienen determinados porque el *COT* no era el esperado en ese paquete, tomando en todos los casos el valor `46`. También se detectaron 4 paquetes mal formados, los cuales no presentaban la información esperada según los datos proporcionados.

Se procesaron 5000000 paquetes.
Ataques: 67
Los cuales el 100 % de ellos fueron TCP: 4893846
Los cuales el 0.000000 % de ellos fueron UDP: 0
Los cuales el 51.683180 % de ellos fueron IEC104: 2584159

IP	Tipo	Cantidad	Porcentaje
XXX.XXX.XXX.XXX	[1]	[2]	0.000283 %
XXX.XXX.XXX.XXX	[1]	[2]	0.000283 %
...	...	...	...
...	...	...	...
...	...	...	...

Los cuales el 14.739600 % de ellos fueron paquetes I: 736980
Los cuales el 6.741980 % de ellos fueron paquetes S: 337099
Los cuales el 30.536520 % de ellos fueron paquetes U: 1526826
TESTFR act: 770447
TESTFR con: 754635
STARTDT act: 1268
STARTDT con: 476
STOPDT act: 0
STOPDT con: 0

Ataques: 67

Malformed:

IP origen	IP destino	Cantidad
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]

Maestro erróneo: 0
El programa ha tardado 1.664274 segundos en ejecutar.
El throughput es de 190319303 Bytes/segundo.

Tabla 5.2: Resultado ejecución con traza de 5.000.000 de paquetes.

También cabe destacar el tiempo de ejecución logrado por el programa, dando por válido la detección de anomalías en tiempo real.

Durante el primer análisis de este archivo, el programa detector identificó un número de anomalías muy superior al esperado, generando un aviso de anomalía por cada paquete con cabecera 104. Gracias a la comprobación con Wireshark se pudo detectar que la traza de red se encontraba dentro de una red virtual, la cual añadía una nueva cabecera usando el formato 802.1Q, el cual es posteriormente redirigido a la dirección de destino esperada[14]. Esto se pudo ver gracias al campo *EtherType*, que son dos octetos correspondientes a la cabecera *Ethernet*. Este, indica el protocolo en el que están encapsulados los datos. En el caso de paquetes con el protocolo tratado en este trabajo, se espera que correspondan al IPv4[4], es decir el valor 0x0800 en hexadecimal, por tanto el programa alertaba de la anomalía en todos los paquetes que no se cumpliera esa condición.

[4]Internet Protocol version 4.

Para solucionar este falso positivo producido por la red virtual, se añadió una cabecera específica para la red virtual. De esta manera, en aquellos casos en los que se contara con la red virtual el programa se comportará de la misma manera que en los casos en los que no estuviera presente, siempre y cuando esta red cuente posteriormente con los datos correspondientes al protocolo esperado, es decir, que el tipo sea IPv4, como se puede apreciar en la **Figura 5.5**.

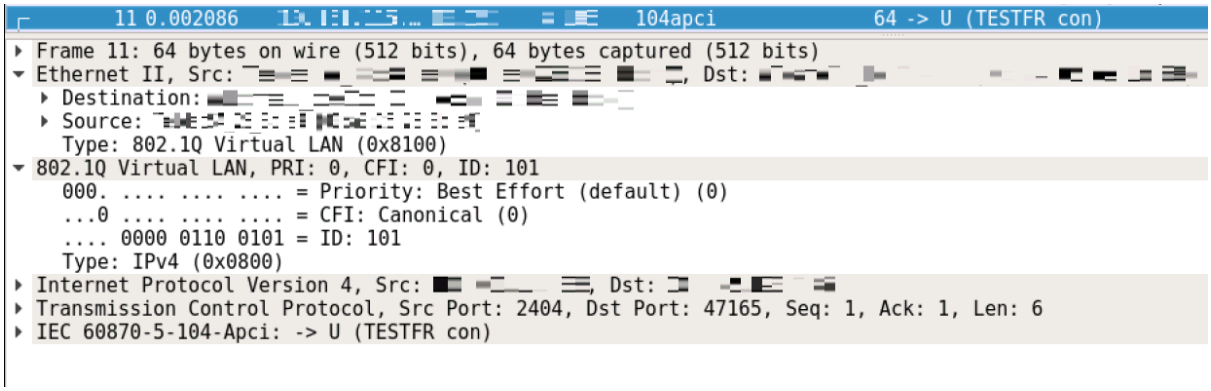


Figura 5.5: Ejemplo de paquete en red virtual.

Tras la presentación de los resultados, se puede observar la clara superioridad en cuanto a tiempos de ejecución que ofrece el detector desarrollado en C en este Trabajo Fin de Grado, frente al desarrollado en Python. En la **Figura 5.6** se muestra una gráfica comparativa en escala logarítmica. Se puede apreciar como el detector desarrollado en Python, posee 4 órdenes de magnitud mayores al detector desarrollado en este trabajo. En ambos programas se ejecutó el mismo archivo de 5.000.000 de paquetes.

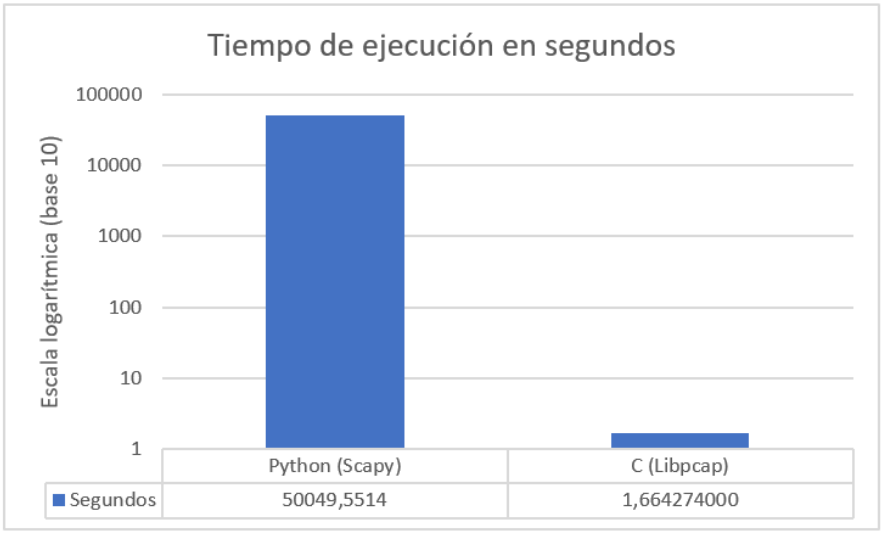


Figura 5.6: Comparación de tiempos de ejecución Python frente a C.

5.6. Conclusión

En este capítulo se ha podido corroborar el correcto funcionamiento del sistema detector, con las diversas comprobaciones realizadas a través de la generación de ataques.

En el apartado 5.2 se ha podido comprobar como se comporta el detector ante tráfico red con protocolo IEC 60870-5-104, ampliándolo más adelante en el apartado 5.5 donde podemos ver como el detector avisa de las anomalías presentes en esa transmisión. También, gracias a la traza facilitada se pudo comprobar la utilidad del sistema usando datos reales y arrojando resultados que podrán ser de utilidad para poder prevenir posibles ataques.

También, se han detallado las grandes diferencias en cuanto a tiempos de ejecución, comparando los resultados obtenidos en este trabajo, frente a los obtenidos en el trabajo de que se precede. Con esta comprobación, se deja claramente plasmado los grandes beneficios que supone el usar C frente a Python para este tipo de sistemas.

Con esto se da por finalizado el detector, dando por satisfecho los resultados mostrados tras la validación.

6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Una vez finalizadas las pruebas de validación y el desarrollo del sistema, queda manifestada la gran eficiencia del detector de anomalías realizado en este Trabajo Fin de Grado. Este proyecto, se encuentra alojado en la página de *Github* `https://github.com/davidsanchesgomez/IEC-60870-5-104` para su consulta.

En este Trabajo Fin de Grado se han explicado las características de las redes SCADA en las que el protocolo IEC 60870-5-104 se encuentra presente. También, se ha explicado en profundidad la estructura de este protocolo de forma que pueda ser de utilidad para posteriores trabajos. Además se hicieron pruebas de validación, de manera que se pudo corroborar el correcto funcionamiento del mismo.

Se puede ver, comparando los resultados del trabajo previo en el que se basa el presente, como los tiempos de ejecución son mucho más funcionales de cara a la implantación de un sistema IDS en tiempo real. Pasando de unos tiempos de análisis de la traza facilitada por la Cátedra UAM-Naudit, usada como medida de comparación, superiores a la hora, a ejecuciones por debajo de dos segundos. Gracias al throughput calculado, también se puede establecer que las velocidades de análisis son superiores al gigabit por segundo. Estos resultados resaltan la clara desventaja de Python frente a C de cara al desarrollo de sistemas en tiempo real, posiblemente causadas por cómo está implementada la biblioteca Scapy usada en el desarrollo con Python.

El desarrollo íntegro de este trabajo se desarrolló en C, motivo por el cual el alumno mejoró sus conocimientos ya obtenidos en asignaturas del Grado como *Programación II*. La depuración de errores usando herramientas como Wireshark, también ayudaron a ampliar los conocimientos de asignaturas como *Arquitectura de Redes I y II*, así como los distintos conceptos de topologías y redes de comunicación, en especial el transporte a través de TCP/IP.

6.2. Trabajo futuro

Una posibilidad de trabajo futuro, podría ser el implementar el detector junto a la captura de tráfico a tiempo real. Se piensa que, haciendo uso de las funciones pcap_open_live junto a pcap_loop, ambas dentro de la librería Libpcap, podría aplicarse dicho cometido.

Otro posible trabajo futuro podría ser hacer un seguimiento de la sesión TCP, a parte de la de los mensajes con protocolo 104. De esta manera, se podría detectar si un paquete es una retransmisión. Con esta detección, se podría parar el análisis de ese paquete, ya que no tendría sentido analizarlo si ya ha sido previamente analizado. Para los paquetes *Out of Order* detectados en *Wireshark*, sería interesante analizar el por qué el programa no lee todos los datos y de esta manera poder establecer un juicio validado, de si es un error del programa o determinar la causa de esta parada de lectura.

También, se podría implementar el código desarrollado, no solo como detector de anomalías, sino también como medidor de rendimientos, ofreciendo datos relevantes sobre el protocolo IEC 60870-5-104 tratado en este trabajo. Esta implementación sería interesante verla en herramientas de análisis de tráfico.

Por último, la optimización del código también sería interesante. La inclusión de mejores ecuaciones para la creación de *key*(claves) para las tablas *hash*, permitirán una disminución de las colisiones en dichas tablas y por lo tanto la reducción de tiempos de ejecución.

Bibliografía

[1] Juan Alberto García Barroso. SCADA el Sistema de Información para Procesos Productivos Industriales. *MoleQla*, 1(33):66–69, 2019.

[2] Petr Matousek. Description and analysis of IEC 104 Protocol. Technical report, Brno University of TechnologyBrno, Czech Republic, Faculty of Information Technology, 12 2017.

[3] Yi Yang, Kieran Mclaughlin, Tim Littler, Sakir Sezer, Bernardi Pranggono, and H F. Wang. Intrusion detection system for iec 60870-5-104 based scada networks. 07 2013.

[4] RocyLuo. Iec104 client and server simulator. `https://github.com/RocyLuo/IEC104TCP`, 2015.

[5] David Abreu Cañamares. Desarrollo de un sitema de detección de tráfico anómalo en redes scada basadas en el protocolo iec 60870-5-104. 2018.

[6] David Cassel. Which programming languages use the least electricity? `https://thenewstack.io/which-programming-languages-use-the-least-electricity/`, 2018.

[7] Miguel. Dcs o plc o pac o rtu? `https://controlreal.com/es/dcs-o-plc-o-pac-o-rtu/`, 2015.

[8] IPCOMM GmbH. Iec 60870-5-104. `https://www.ipcomm.de/protocol/IEC104/en/sheet.html`, 2017.

[9] davideto6 David Abreu Cañamares. Desarrollo de un sistema de detección de tráfico anómalo en redes scada basadas en el protocolo iec 60870-5-104. `https://github.com/davideto6/IEC-60870-5-104`, 2018.

[10] Carlos Pacho Marshall. Iec 60870-5-104 protocol detection rules. `https://blog.snort.org/2016/12/iec60870-5-104-protocol-detection-rules.html`, 2016.

[11] Craig Leres Van Jacobson and Steven McCanne. Reference manual pages (3pcap). `https://www.tcpdump.org/manpages/pcap.3pcap.html`, 2018.

[12] Álvaro Culebras Sánchez. Desarrollo de un sistema de monitorización de redes scada para la detección de tráfico anómalo. `https://repositorio.uam.es/bitstream/handle/10486/669715/Culebras_Sanchez_Alvaro_tfg.pdf?sequence=1&isAllowed=y`, 2016.

[13] automayt Jason Smith. Ics-pcap.
`https://github.com/automayt/ICS-pcap/tree/master/IEC%2060870/iec104`, 2016.

[14] Margaret Rouse. The voip basics every enterprise should know. definition: Vlan (virtual lan).
`https://searchnetworking.techtarget.com/definition/virtual-LAN`, 2018.

A

Apendice

A.1. Resultado de la ejecución de la traza facilitada de 5.000.000 de paquetes. Se obvian las IP por privacidad.

NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 69789
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 148466
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 226605
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 305807
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 385331
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 464555
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 544364
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 622953
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 702464
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 782132
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 861767
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 941349
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1020917
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1100351
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1179565
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1259495
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1339890
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1420219
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1501182
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1582068
NEW ATTACK!. Malformed Packet. Objects size 8 doesn't match with objects type 4 at packet 1621370
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1660605

NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1739392
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1819466
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1820462
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1899541
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 1980022
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2061454
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2141319
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2221550
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2301598
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2380694
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2460766
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2541680
NEW ATTACK!. Malformed Packet. Objects size 8 doesn't match with objects type 4 at packet 2548302
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2621174
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2700939
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2781396
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2861879
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 2942137
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3023079
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3102126
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3181333
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3262484
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3342119
NEW ATTACK!. Malformed Packet. Objects size 8 doesn't match with objects type 4 at packet 3362709
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3421036
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3500836
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3580407
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3659278
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3739821
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3820081
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3899420
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 3980694
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4060394
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4140649
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4221859
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4302021
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4381306
NEW ATTACK!. Malformed Packet. Objects size 8 doesn't match with objects type 4 at packet 4416452
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4461670
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4541002
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4621183
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4701853
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4780420
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4860689
NEW ATTACK!. Wrong Cause of Transmission: 46 with type: 100 at packet: 4942817

Se procesaron 5000000 paquetes.

Ataques: 67
Los cuales el 100 % de ellos fueron TCP: 5000000
Los cuales el 0.000000 % de ellos fueron UDP: 0

Los cuales el 51.683180 % de ellos fueron IEC104: 2584159

Los cuales el 14.739600 % de ellos fueron paquetes I: 736980

IP	Tipo	Cantidad	Porcentaje
XXX.XXX.XXX.XXX	[1]	[2]	0.000283 %
XXX.XXX.XXX.XXX	[1]	[2]	0.000283 %
...	...	...	...

Los cuales el 14.739600 % de ellos fueron paquetes I: 736980

Los cuales el 6.741980 % de ellos fueron paquetes S: 337099

Los cuales el 30.536520 % de ellos fueron paquetes U: 1526826
TESTFR act: 770447
TESTFR con: 754635
STARTDT act: 1268
STARTDT con: 476
STOPDT act: 0
STOPDT con: 0

Los cuales el 51.683180 % de ellos fueron IEC104: 2584159

Ataques: 67

Malformed:

IP origen	IP destino	Cantidad
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]
XXX.XXX.XXX.XXX	XXX.XXX.XXX.XXX	[1]

Maestro erróneo: 0
El programa ha tardado 1.664274 segundos en ejecutar.

El throughput es de 190319303 Bytes/segundo.